Policy Segmentation for Intelligent Firewall Testing

Adel El-Atawy, Khaled Ibrahim, Hazem Hamed, Ehab Al-Shaer

School of Computer Science, Telecommunication, and Information Systems
DePaul University, Chicago, Illinois, USA
Email: {aelatawy,kibrahim,hhamed,ehab}@cs.depaul.edu

Abstract— **Firewall development and implementation are constantly being improved to accommodate higher security and performance standards. Using reliable yet practical techniques for testing new packet filtering algorithms and firewall design implementations from a functionality point of view becomes necessary to assure the required security. In this paper, an efficient paradigm for automated testing of firewalls with respect to their internal implementation and security policies is proposed. Randomly testing the firewall matching functionality requires exponential number of testing scenarios and thus an impractically long testing period. We propose a novel firewall testing technique using policy-based segmentation of the traffic address space, which can intelligently adapt the test traffic generation to target potential erroneous regions in the firewall input space. We also show that our automated approach of test case generation, analyzing firewall logs and creating testing reports not only makes the problem solvable but also offers a significantly higher degree of confidence than random testing.**

I. Introduction

In enterprise networks, the security of the internal network against attacks, illegitimate traffic, and unauthorized access can be a crucial to the success of the entire business operation. As firewalls are core elements in network security, deploying correctly functioning firewalls is mandatory to accomplish these security goals. Firewalls are filtering devices usually placed at the boundary of the network to block unwanted traffic from/to external networks, as well as to regulate traffic flow between domains inside the same network [8]. Firewalls usually undergo continuous modification to their internal design and implementation to improve performance [10] or to add new features [9] (*e.g.*, extra filtering fields, enhanced logging, etc.) Thus, the firewall matching functionality always needs to be tested and verified to have some assurance on the correctness of the firewall implementations.

The complete framework of firewall testing should contain two components: (1) generating test packets that test the firewall given a certain policy, and (2) generating various policies scenarios to test the firewall handling of different policy styles/configuration. Although both are needed to claim a complete testing environment for firewalls, our focus in this paper is on the first problem, *i.e.*, given a firewall/policy, how to ensure that the firewall implements this policy configuration correctly.

Testing the firewall by exhaustively injecting all possible packets into the firewall will be enough. However, this is infeasible due to the huge number of packets needed. Even if we try to restrict the test traffic to the range of relevant destination addresses only, it is still an impossible task to do. With destination address limitation and using many optimizations, the required testing time can be reduced from about 4×10^{13} years to 4.5 years at a test rate of 1G packets/second. Both traffic and time needed are still prohibitive. Random sampling/testing can also be used but its one-sided error probability (*i.e.*, probability of faulty firewall passes the test) is impractically high.

A smart method to select the packets used for testing is needed to save time and increase the test accuracy. Using a careful study of the policy, its rules, and the inter-rule interactions; the different decision paths of the firewall filtering algorithm can be tested individually. The whole range of different packet header values (*i.e.*, protocol, source address/port, destination address/port) is divided into different regions (segments) based on the policy. Each segment is tested according to some measures (weight) that reflects the important of this segment in testing in order to achieve the best testing accuracy. The formalization, implementation and evaluation of this framework are discussed in this paper.

In the next section, a quick overview of related work is provided. Section III discusses the system framework and its different modules. The test data generator is discussed in Section IV. In Section V the system is evaluated. Finally, Section VI presents the conclusion and plans for future work.

II. Related Work

Testing is considered not only one of the most important phases but also the most expensive and time consuming phase in any production line. Testing techniques are categorized into two main techniques: black-box testing (or functional testing) and white-box testing (or structural testing) [2], [6]. In black-box testing, the internal structure or implementation of the component under testing is unknown. In this approach, test case generation is performed either exhaustively or selectively based on statistical techniques. On the other hand, the white-box testing approach assumes the knowledge of the internal structure or the implementation to generate the test cases. In the context of our work in this paper, the black-box testing approach is more appropriate for testing firewall as the filtering/matching implementation details may not be known.

We can generally categorize firewall testing into two groups: *offline* and *online* testing techniques. In offline testing, simulation is used to mimic the network and firewall behavior and discover configuration errors. In [11], a CASE tool is used to

model the firewall and the surrounding networks in order to generate firewall test cases against a list of simulated security vulnerabilities. On the other hand, in [16], [1] a firewall analyzer was presented to simulate firewall behavior given a certain policy configuration. It also allows for designing user-defined queries and test cases to examine the firewall operation. In these approaches, the simulation is performed totally offline without injecting any packets into the firewall.

In the online testing approach, real traffic is injected into the network and the corresponding firewall behavior is then examined. The currently available online testing techniques are mainly focused on performing firewall penetration tests, where a set of vulnerability scanners are configured to generate malicious traffic against the firewall. In [4], the authors presented a methodology to test the firewall security vulnerability using two tests one is automated using a security analysis tool (SATAN) while, the other test is manual which is based on interacting with and observing the firewall. In [12], policy-oriented approach is used to test the firewall against a list of predefined vulnerability using various policies and hardness levels. A functional approach is presented by in [13], [15] where firewall vulnerabilities are classified based on the firewall functional units to run the corresponding penetration tests for different firewall products. A different approach was taken in [14] by deliberately introducing errors in the software modules of IPChains and NetFilter firewalls, and examined the security vulnerabilities caused by these errors. In the CERT guide to system and network security [5], experts recommend a firewall testing strategy in which the boundaries of packet filter rules are identified and then test samples are generated from the regions immediately adjacent to each boundary. Although this looks to be the closest one to our approach, no formalization or specific technique was described.

In conclusion, most of the related work is focused on designing penetration testing using predefined test cases or based on network information and queries. We consider our work novel as it is the first to consider the policy structure and semantics for constructing the most critical test cases avoiding random and exhaustive testing.

III. Testing Framework Overview

The testing framework consists of the following components: segmentation module, segment weight analyzer, test packet generator, and test analyzer (see Fig. 1). The following is the description of each component.

- Segmentation module: This component converts the device-specific policies to a canonical policy format and segments them based on rule interactions [3] and network information.
- *Segment weight analyzer*: In this component, the segments are analyzed and each segment is assigned a weight based on a number of factors such as the number of rules intersecting in the segment and the size of the segment address space. The segment weight determines the testing intensity (number of generated test packets) for this segment.

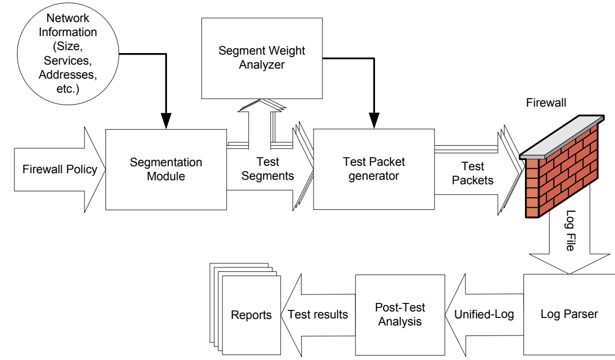


Fig. 1. Firewall testing framework

- *Test packet generator*: This component generates test packets distributed in proportion to the segment weight. For example, segments that involve many rules in the policy and/or heavily populated with assigned network addresses will potentially receive more test packets than others. Each test packet carries test case information that includes a packet sequence number, the targeted segment id and the expected filtering action.
- *Test analyzer*: As a result of the firewall test cases, the firewall logs the matching results, which are then parsed and analyzed to validate the firewall operation. Using the test case information (embedded in each packet), the test analyzer matches the log results with the test case information to determine erroneous cases. In order to make this module device independent, the firewall log is converted to a unified format before the final analysis. The generated reports for each test case are stored in a repository for further inquiries and analysis.

IV. Adaptive Smart Traffic Generation

There are three different approaches to generate test packets. (1) The *exhaustive* approach, in which we generate all possible packets, which requires a prohibitively large number of generated packets as well as long analysis time. (2) The *random* approach, in which, we select randomly a set of packets that will hopefully represent the different cases of the algorithm and hit an error if any exists. This mode generates a fewer number of packets but the probability of missing some of the decision paths of the filtering algorithm is quite high, as well as we do not differentiate between the importance of different regions of the filtering algorithm's input space. (3) The *selective* mode, in which we consider the differences between the different segments of the input space, and we can intelligently test the various decision paths of the algorithm.

In this section, we describe our selective testing approach in details. Our main goal is to generate the least amount of traffic that is needed to test all the possible decision paths for the given firewall policy. In order to have this span over the firewall/policy behavioral cases, we will have to define the space over which the entire policy could be tested.

Definition 1: The *traffic address space* is the space whose elements are all the different tuples that identify traffic flows in a network environment, especially from the firewall point of view. This is usually composed of the transport protocol, source address/port, and destination address/port. However, other fields could also be used such as IP TTL and flags.

A. Firewall rule representation

Rules and address spaces are represented as Boolean expressions where a packet satisfies this expression if and only if its header information is contained in this address space. We use Binary Decision Diagrams (BDD's) as a standard canonical method to represent Boolean expressions [7].
Each firewall rule is composed of filtering fields, namely:
<protocol, source IP, source port, destin. IP, destin. port>
The rule Boolean expression is created by concatenating the binary representation of all rule fields into one chunk of bits. Bits in this expression can be a value (0 or 1) or don't care. Each bit of this chunk is assigned a Boolean variable, and for each rule either the variable or its complement is taken into the expression, depending on the value of the corresponding bit in the rule. For example, consider this rule:

R = <any, 67.*.*.*, any, 121.*.*.*, any>

The corresponding Boolean expression (ϕ) includes only the non-don't care variables for the source IP (bits x_4 to x_{35}) and the destination IP (bits x_{53} to x_{83}). Therefore:

$$\phi = (x_4' \wedge x_5 \wedge x_6' \wedge x_7' \wedge x_8' \wedge x_9' \wedge x_{10} \wedge x_{11}) \\ \wedge (x_{53}' \wedge x_{54} \wedge x_{55} \wedge x_{56} \wedge x_{57} \wedge x_{58}' \wedge x_{59}' \wedge x_{60})$$

As we see, 100 variables were reduced to 16 variables. This is a common case, where rules in policies are normally aggregates, and a small percentage of them use specific values for all its fields. In this paper we refer to the mapping from the firewall rule R_i into the Boolean expression ϕ_i by the following function:

$$\phi_i = AS(R_i)$$

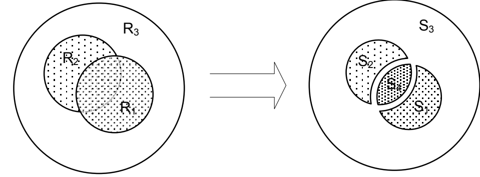
B. Policy-based address space segmentation

Our goal here is to investigate the behavior as thoroughly as possible while keeping the testing traffic volume at a minimum. Thus, we need to determine all interactions exhibited in the firewall policy. In order to test every possible behavioral pattern (*i.e.*, decision path) of a firewall individually and efficiently, we selectively generate a small set of test packets tailored for each decision path. These decision paths can be determined by intersecting all the address spaces of the rules.

Definition 2: A *segment* is a subset of the total traffic address space that belongs to one or more rules, such that each of the member elements (*i.e.*, header tuples) conforms to exactly the same set of policy rules.

In other words, packets that belong to the same segment are identical from the point of view of the policy. It is important to note that (1) all segments are pairwise disjoint, and (2) any segment can be uniquely identified via the rules that apply

R1:	tcp	121.63.*.*: any	143.91.78.*: any	accept
R2:	tcp	121.63.71.*: any	143.91.*.*: any	accept
R3:	any	*.*.*.*: any	*.*.*.*: any	deny



(a) Policy address-space (b) Segmented address-space

Segment	AS	R_{in}	R_{eff}	OR	ACT
S_1	ϕ_1	$\{R_1, R_3\}$	$\{R_1, R_2, R_3\}$	R_1	accept
S_2	ϕ_2	$\{R_2, R_3\}$	$\{R_1, R_2, R_3\}$	R_2	accept
S_3	ϕ_3	$\{R_3\}$	$\{R_1, R_2, R_3\}$	R_3	deny
S_4	ϕ_4	$\{R_1, R_2, R_3\}$	$\{R_1, R_2, R_3\}$	R_1	accept

Fig. 2. Policy-based address space segmentation example.

to it. The first property shows that there is no redundancy when segments are tested individually and the second property ensures that testing all segments implies testing all different decision paths with respect to the policy used.

To completely identify a segment, each one is associated with the following information:

- *Address space (AS):* The Boolean expression representing the address space of the segment.
- *Included rules (R_{in}):* Ordered list of rules (*i.e.*, as ordered in the policy) containing this segment.
- *Excluded rules (R_{out}):* Ordered list of rules not covering this segment (*i.e.*, complement of R_{in}; $R - R_{in}$).
- *Effective rules (R_{eff}):* Ordered list of rules that contribute to the formation of the segment.
- *Owner rule (OR):* The first rule in R_{in} will be taken as the owner of the segment.
- *Filtering action (ACT):* Firewall action to be taken for this segment. This is taken as the action of the first rule in the R_{in} list (*i.e.*, the owner rule).

Fig. 2 shows a segmentation example of a simple firewall policy composed of three rules: R_1, R_2 and R_3. As a result of intersecting the three rules, four address segments are produced: S_1, S_2, S_3 and S_4. The information associated with each segment is presented in the table in Fig. 2.

The segmentation algorithm (Algorithm 1) takes as input the policy rules (R), the default action of the firewall (defAct), and the initial domain (InitDomain) which is the total traffic address space under consideration such as the network address space or the default address space. More discussion of this issue can be found in section IV-C. As a result it outputs the list of segments induced by the given policy.

At the beginning, the initial segment is initialized, and added to SEGLIST (the list holding the whole list of segments created so far) at lines 1 and 2. Then we loop over the rules to impose their effect on all the existing segments. We use three variables just for the sake of readability of the algorithm; S, $IncSeg$, and $ExcSeg$ to refer to the currently processed

Algorithm 1 DoSegmentation (R, defAct, InitDomain)

1: $SEGLIST \leftarrow \Lambda$
2: AddSegment (InitDomain,Λ ,Λ , defAct)
3: **for all** rules: i = 1 to n **do**
4: **for** segments: j = $SEGLIST.Count$ downto 1 **do**
5: $S = SEGLIST[j]$
6: $IncSeg \leftarrow S.AS \wedge AS(R_i)$ {Included part of the segment}
7: $ExcSeg \leftarrow S.AS \wedge \neg AS(R_i)$ {Excluded part of the segment}
8: **if** $IncSeg \neq Seg.AS$ **then** {Segment not contained in the Rule's AS}
9: **if** $IncSeg \neq \Phi$ **then**
10: AddSegment ($IncSeg$, $S.R_{in} \cup \{R_i\}$, $S.R_{out}$, $S.R_{eff} \cup \{R_i\}$)
11: AddSegment ($ExcSeg$, $S.R_{in}$, $S.R_{out} \cup \{R_i\}$, $S.R_{eff} \cup \{R_i\}$)
12: **else** {no intersection between the rule and the segment}
13: AddSegment ($ExcSeg$, $S.R_{in}$, $S.R_{out} \cup \{R_i\}$, $S.R_{eff} \cup \{R_i\}$)
14: **end if**
15: **else** {Segment is inside the Rule's AS}
16: AddSegment ($IncSeg$, $S.R_{in} \cup \{R_i\}$, $S.R_{out}$, $S.R_{eff}$)
17: **end if**
18: SEGLIST.Delete (Segment j) {delete the original segment}
19: **end for**
20: **end for**
21: **return** SEGLIST

segment, the overlapping portion of the segment and the rule's address space, and the non-overlapping portion (*i.e.*, remaining address space of the segment), respectively.

We have three cases of interaction between the segment and the rule's address space; (1) either split the segment into included and excluded area, (2) leave the segment space intact as an excluded segment if the rule doesn't intersect with this rule, or (3) leave the segment space unmodified as an included space if the rule is a superset of the segment.

Restricting the segment adding (AddSegment) to non-empty ones (lines 8, and 9) is necessary; otherwise exponential running time in the number of policy rules is guaranteed as there will be a deterministic growth in the number of segments by doubling the number of segments after each rule, resulting in a list of 2^n segments. It is also important to notice that the format constraints of firewall rules such as using contiguous address spaces and a limited range of field values significantly reduces the complexity of the segmentation algorithm, which, in the general case, could yield exponential running time.

AddSegment is used to add a new segment to the total segment list. If R_{in} is an empty list, the action (ACT) of the segment is set to the default (defAct), otherwise it takes the action used in the first rule (assuming rule priorities are their order in the policy). Similarly, if the R_{in} is non-empty, the first rule is taken as the owner rule (OR).

C. Choosing the initial address space

There are three ways to set the value of the initial address space (InitDomain). First, the initial domain can be the entire possible traffic address space (*i.e.*, corresponding to: <any, *.*.*.*, any, *.*.*.*, any>). Second, we can use the traffic address space corresponding to <any, *.*.*.*, any, "network range", any>. This can also be extended by ORing the AS of the network range with the AS of the desired additional address space such as multicast traffic (*i.e.*, <UDP, *.*.*.*, any, 224.*.*.*/4, any>). They differ in the address range used and the desire to restrict the testing scope. Third, $InitDomain$ can be restricted to the address space used (*i.e.*, assigned to hosts and active services) in the network in order to further reduce the testing time. However, this limits the testing coverage and accuracy.
To combine the advantages of these options, we consider each one with different testing intensity based on the density of the address space used in the resulting segments such that the segments including more used network addresses will have more testing samples.

D. Measuring the segment weights

Having a measure for the importance (weight) of each segment is essential, as this leads us to decide how intense the testing should be within a certain segment. The weight of a segment is mainly determined by its inherent properties that might impact the filtering algorithm decision. The segment weights can also be adjusted by the administrator if necessary to reflect the business needs such as when a segment represents critical domains or servers. However, our focus in this section will be on the segment properties and how they affect its weight. From the testing point of view, importance is a measure of the probability that an error can arise in this segment (*i.e.*, a packet is treated differently from what stated in the policy). We identify the following factors that affect the segment weight:

1) *Number of rules intersecting in the segment ($|R_{in}|$):* As the number of rules increases within a segment, the filtering algorithm will face more decisions to make in order to find which rule should be applied. The more decisions the higher the potential of making errors. This factor is normalized by $|R|$; the number of rules in the policy.
2) *Number of effective rules ($|R_{eff}|$):* When a rule intersects non-trivially with a segment splitting it into two non-empty segments, this rule becomes a member of the set of effective rules of this segment. Similarly, as the number of rules in this list increases, the matching algorithm will have more decisions to make in order to reach this segment, thus increasing the error potential. This factor is also normalized by $|R|$.
3) *Weight of the owner rule ($wt(S.OR)$):* The owner rule of a segment is considered important because, effectively, packets match this segment only if they match this rule. Thus, in correct firewall implementations, the action of a segment is exclusively determined by the owner rule. In general, the rule weight ($wt(r)$) is determined by (1) the complexity of the rule as presented in the rule format (*e.g.*, basic vs. extended [9]), (2) the value of the fields (*i.e.*, specific values are more important than wildcards), and (3) the number of related rules preceding this rule.

4) *Cumulative weights of all contributing rules:* The weights of all the rules involved in a segment (*i.e.*, $R_{in} \cup R_{eff}$) can affect the complexity of the matching decision for this segment, causing a higher probability of making errors in the matching. This is because matching a packet within a segment can involve the evaluation of all the involved rules before applying their priority to choose the final rule (the owner rule if a correct decision is reached).
5) *Segment address space size:* The main purpose of this metric is to allow generating more test traffic for segments that include more actually used addresses in the network because these segments are more critical for this particular network. This means that the testing intensity increases proportionally with the ratio of the used address space to the total allocated address space in the segment.

We formalize the testing intensity ρ_i of a segment S_i as a function that considers of all the factors discussed above as follows:

$$\rho_i = w_1 \frac{|S.R_{in}| + |S.R_{eff}|}{|R|} + w_2.wt(S.OR) + w_3 \sum_{r \in R_{in} \cup R_{eff}} wt(r) + w_4 \frac{\|S.AS_{used}\|}{\|S.AS\|}$$

In this formula, each factor is given a weight ($w_1 \dots w_4$) to enable tuning the segment-weight calculation function with the filtering algorithm under test. For example, when linear rule matching is used, more emphasis should be given to $wt(S.OR)$, so a possible weight assignment is (w_1=0.1, w_2=0.7, w_3=0.1, w_4=0.1). On the other hand, rule/field cross-producting algorithms [10] use overlapping sets of rules, and therefore more weight should be given to $|R_{in}|$ and $|R_{eff}|$, for example (w_1=0.4, w_2=0.1, w_3=0.4, w_4=0.1). If the underlying filtering algorithm is unknown, we recommend using equal weight values for all of these factors.

When test packets are generated, the packet header information is distributed on the policy address space in proportion to the calculated segment weights. During a test interval of T seconds and using a rate of R packets/second, the number of generated packets n_i for segment S_i is given by the formula:

$$n_i = \frac{T.R.\rho_i}{\sum_{S_j \in S} \rho_j}$$

E. Post-test analysis

The post-test analysis is the process through which we can detect whether the firewall is functioning correctly or not through analyzing the testing results from the log. The order of the generated packets in the test cases files is the same as their order in the U-Log file. To keep track of the current packet order we will maintain a line counter LC.

Algorithm 2 processes the log to detect two possible types of errors. The first, is when the expected matching action (EMA) is different than the observed matching action (OMA) (Lines 5-7), while the second is when there is a malfunctioning device operation causing missing (Lines 15,16) or duplicate (Lines 17,18) log entries.

Algorithm 2 Post Test analysis

```
1: LC = 0, LSN = 0, CSN = 0;
2: while not end of the U-Log file do
3:    EMA= Expected Matching Action
4:    OMA= Observed Matching Action
5:    if OMA ≠ EMA then
6:       Call Error1 (LC);
7:    end if
8:    if LSN = 0 then {a new U-LOG file or SN reset}
9:       CSN ←Current packet Sequence Number
10:      LSN ← CSN − 1
11:   else
12:      LSN ← CSN
13:      CSN ←Current packet Sequence Number
14:   end if
15:   if CSN > LSN + 1 then {packet is missed}
16:      Call Error2 (LC, CSN, LSN);
17:   else if CSN < LSN + 1 then {packet is duplicated}
18:      Call Error3 (LC, CSN, LSN);
19:   end if
20:   LC ← LC + (CSN − LSN)
21: end while
```

V. Evaluation and Results

As described in Section IV-D, the segment weight function was formalized in order to reflect a strong correlation with the probability of firewall implementation errors. In order to evaluate the efficiency of our testing framework, we investigate the effect of various parameters such as the degree of segment weight-fault correlation, policy style (*i.e.*, the inter-relation between the rules within the policy [3]), and segment sizes, and then compare the accuracy of our scheme with the random test generator. The random tester generates packets uniformly over the whole policy address space within a given time period.

Fig. 3-(a) shows the effect of segment fault probability on the accuracy of our presented approach compared to the random approach using a full range of weight-error correlations. Our approach shows a significant improvement over the random tester even when zero correlation is used. This is attributed to the policy segmentation technique that ensures that each segment will be tested. This is unlike the random tester where entire segments might be skipped.

Fig. 3-(b) shows that our system gives significantly better results over the random sampling in all cases when the policy style is investigated. However, there is a general tendency that policies with high interaction and/or high portion of specific rules (*e.g.*, where all tuples are specified) would give better performance with our technique rather than with the random sampling approach.

Notice that in both Fig. 3-(a) and (b) we took a conservative approach by neglecting the tiny segments in order to remove their biasing effect. However, segments with very small address spaces are usually tested exhaustively, thus causing our technique to be superior over random testing but this might hide the effect of correlation and policy style.

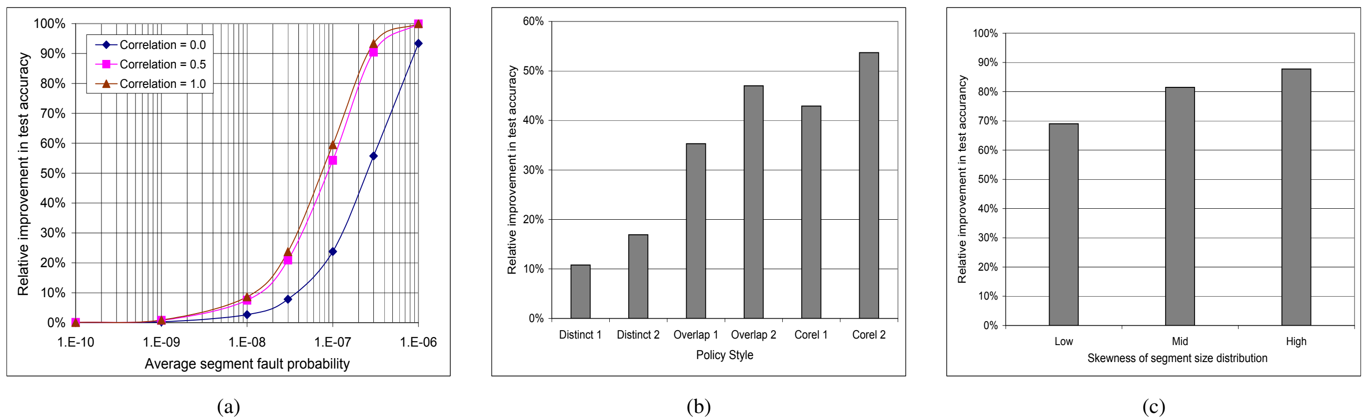


Fig. 3. Evaluation of different factors affecting the improvement in the firewall test accuracy relative to random-sampling test.

Fig. 3-(c) shows the effect of the segment size distribution on the accuracy of our approach. It clearly indicates that the more the distribution is skewed the better the accuracy of the segmentation-based policy testing scheme. However, even with low skewness, the improvement is shown to be as a high as 70% over random testing.

In conclusion, the segmentation based technique shows significant superiority over random sampling under various conditions. Moreover, when using more test options like exhaustive testing of small segments, and utilizing network information, the testing results become orders of magnitude more accurate within reasonable testing times.

VI. Conclusion and Future Work

This paper presents a new efficient testing technique for the implementation of filtering-based security devices such as firewalls. The technique avoids both the exhaustive and random testing via segmenting and classifying the firewall policy address space based on critical testing factors such as the rule interaction, order, complexity, and network information. Using this approach, our evaluation study shows significantly better accuracy and performance than random testing as the probability of missing all fault locations becomes considerably small. Our approach is also shown to be robust as it maintains better results than random sampling even when there is a small correlation between estimated segment weight and the probability of error/fault. When policies of various styles and different segment sizes are used in our evaluation study, our approach is shown to be superior in all cases. It is also shown that the policy segmentation approach has more advantage (as high as 70-90% over the random testing) as rule interaction and/or the skewness of the segment size distribution increases.

Currently, our research is in-progress to bundle the system with a policy generator module in order to test a firewall under several scenarios of policy configurations. Taking into consideration that policies should be as orthogonal as possible in their relation with the filtering algorithm, renders the selection of efficient test policies a really hard problem. Studying the segmentation behavior for several policy styles needs further investigation. More refinement to the weight function to consider various filtering techniques used in IPSec and IDS devices is also under investigation.

References

[1] A. Wool A. Mayer and E. Ziskind. Fang: A Firewall Analysis Engine. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P'2000)*, pages 85–97, August 2000.

[2] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, Verification, and Testing of Computer Software. *ACM Computer Survey*, 14(2):159–192, 1982.

[3] E. Al-Shaer and H. Hamed. Modeling and Management of Firewall Policies. *IEEE Transactions on Network and Service Management*, 1(1), 2004.

[4] K. Al-Tawil and I. Al-Kaltham. Evaluation and Testing of Internet Firewalls. *International Journal of Network Management*, 9(3):135–149, 1999.

[5] J. Allen. *The CERT Guide To System and Network Security Practices*. Addison-Wesley, 2001.

[6] Boris Beizer. *Black-Box Testing Techniques for Functional Testing of Software and Systems*. Wiley-VCH, 1995.

[7] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[8] D. Chapman and E. Zwicky. *Building Internet Firewalls, Second Edition*. Orielly & Associates Inc., 2000.

[9] R. Deal. *Cisco Router Firewall Security*. Cisco Press, 2004.

[10] P. Gupta and N. McKeown. Algorithms for Packet Classification. *IEEE Network*, 15(2), 2001.

[11] J. Jürjens and G. Wimmel. Specification-Based Testing of Firewalls. In *Proceedings of the 4th International Conference on Perspectives of System Informatics (PSI'02)*, pages 308–316, 2001.

[12] M. Lyu and L. Lau. Firewall Security: Policies, Testing, and Performance Evaluation. In *Proceedings 2000 International Conference on Computer Systems and Applications (COMPSAC'2000)*, pages 116–121, October 2000.

[13] E. Schultz M. Frantzen, F. Kerschbaum and S. Fahmy. A Framework for Understanding Vulnerabilities in Firewalls Using a Dataflow Model of Firewall Internals. *Computers and Security*, 20(3):263–270, May 2001.

[14] Z. Kalbarczyk R. Iyer S. Chen, J. Xu and K. Whisnant. Modeling and Evaluating the Security Threats of Transient Errors in Firewall Software. *Performance Evaluation*, 56(1-4), 2004.

[15] E. Schultz F. Kerschbaum S. Kamara, S. Fahmy and M. Frantzen. Analysis of Vulnerabilities in Internet Firewalls. *Computers and Security*, 22(3):214–232, April 2003.

[16] A. Wool. Architecting the Lumeta Firewall Analyzer. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.